INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING
Int. J. Numer. Meth. Engng 2012; **89**:1635–1651
Published online 12 December 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/nme.3302

OpenCL-based implementation of an unstructured edge-based finite element convection-diffusion solver on graphics hardware

F. Mossaiby[1,*,†], R. Rossi[2,3], P. Dadvand[2,3] and S. Idelsohn[2,4]

[1]*Department of Civil Engineering, Faculty of Engineering, University of Isfahan, 81744-73441 Isfahan, Iran*
[2]*Centre Internacional de Mètodes Numèrics en Enginyeria (CIMNE), Barcelona, Spain*
[3]*UPC, BarcelonaTech, Campus Norte UPC, 08034 Barcelona, Spain*
[4]*ICREA Research Professor at CIMNE, Barcelona, Spain*

SUMMARY

The solution of problems in computational fluid dynamics (CFD) represents a classical field for the application of advanced numerical methods. Many different approaches were developed over the years to address CFD applications. Good examples are finite volumes, finite differences (FD), and finite elements (FE) but also newer approaches such as the lattice-Boltzmann (LB), smooth particle hydrodynamics or the particle finite element method. FD and LB methods on regular grids are known to be superior in terms of raw computing speed, but using such regular discretization represents an important limitation in dealing with complex geometries. Here, we concentrate on unstructured approaches which are less common in the GPU world. We employ a nonstandard FE approach which leverages an optimized edge-based data structure allowing a highly parallel implementation. Such technique is applied to the 'convection-diffusion' problem, which is often considered as a first step towards CFD because of similarities to the nonconservative form of the Navier–Stokes equations. In this regard, an existing highly optimized parallel OpenMP solver is ported to graphics hardware based on the OpenCL platform. The optimizations performed are discussed in detail. A number of benchmarks prove that the GPU-accelerated OpenCL code consistently outperforms the OpenMP version. Copyright © 2011 John Wiley & Sons, Ltd.

Received 21 January 2011; Revised 8 August 2011; Accepted 10 August 2011

KEY WORDS: unstructured grids; convection diffusion; edge-based; GPU; OpenCL; OpenMP

1. INTRODUCTION

Multicore systems with increasing computational capabilities are widely available in design offices. This fact together with the advances in the numerical methods in computational fluid dynamics (CFD) allows the solution of increasingly complex problems. The limitations in CPU hardware design, however, seem to indicate that the speed of the single core systems is not likely to increase over the next years. In contrast, the future appears to open the way to the availability of multicore and many-core processors even on commodity systems. A relatively new player in this context is represented by programmable graphical processing units (GPUs) which offer outstanding computational power at a comparably cheaper price point. Over the last years, a number of different applications were designed to incorporate the computational power of GPUs. These codes were based on very different software platforms. Early developers had to cast their algorithms in terms of graphics operations in order to use the computational power of GPUs [1, 2]. The interface used was usually general graphics libraries, OpenGL. These developers had to face many limitations such

*Correspondence to: F. Mossaiby, Department of Civil Engineering, Faculty of Engineering, University of Isfahan, 81744-73441 Isfahan, Iran.
†E-mail: mossaiby@eng.ui.ac.ir

as the low precision of the floating point operations. As GPUs were designed to perform specific graphics operations, the extension of the code to general cases was often difficult or impossible. A survey of general purpose computation on graphics hardware can be found in the excellent work of Owens *et al.* [3].

With the growing demand towards general processing on graphics processing units (GPGPU), GPU vendors started to develop platforms especially designed for this purpose. The most prominent example of these software platforms is NVIDIA's CUDA [4] dominating the GPGPU market since its introduction in the middle of February 2007. Unfortunately these platforms were designed to work exclusively on their vendor's hardware, which made the codes unportable. In December of 2008, the open computing language (OpenCL) was announced by a group of hardware and software companies. OpenCL was designed as an open standard for developing high-performance programs on a wide range of hardware platforms, such as CPUs and GPUs. This open standard provides an alternative to proprietary software platforms. The specific design objective of OpenCL was thus, for the first time, to provide the possibility of writing portable code on a variety of different hardware. Although this is very promising, it is noteworthy that portability of optimizations on different hardware platforms is still an open issue, because of a combination of immaturity of the current software platforms and of inherent design differences in the hardware. Since the announcement of OpenCL, many vendors including AMD, NVIDIA and Intel have started to provide support for this new standard. Our effort in the current work is to define the basis for a *portable framework* for CFD-like computations using the finite element (FE) method. We should remark that portability is intended here in the sense that the code should run unmodified on any target architecture; specialized performance tuning may however be needed to obtain optimal performance on different architectures. This requirement implies choosing OpenCL as the only open standard to guarantee compatibility between multiple architectures.

From a conceptual point of view, many different methods were developed over the years to deal with problems in CFD. Finite difference (FD) type solvers on regular grids represent the most well-established technique and are known for their high computational efficiency and low memory usage that allowed them dealing with very fine meshes. Similarly, typical lattice-Boltzmann (LB) techniques assume a form that is largely similar to FD schemes and were shown in recent years to be well suited for the solution of incompressible flow problems. The disadvantage of such methods is however typically related to the need of using regular discretizations of the space, which constitutes a major drawback in dealing with complex geometries. Finite volume schemes on the other hand inherit some of the characteristics of FD schemes but allow their extension to nonstructured discretizations. A different but equally important approach is constituted by the FE method, which provides a solid mathematical basis and allows dealing with a large variety of problems arising in CFD (see [5–9] for some examples of application on fixed or changing meshes). An important advantage of the FE over alternatives is the natural ability to deal with arbitrarily shaped domains, through the use of unstructured meshes.

Given its simplicity and high level of parallelism, the LB method is suited for the implementation on GPUs, and indeed a number of different solvers were developed both in CUDA and more recently in OpenCL. Similarly, a number of CUDA-based efforts targeted to the development of unstructured FE solvers were proposed recently. GPUs were proved to be very effective for the implementation of DG methods, see [10], due to the very high arithmetic intensity of DG. Alternatively, general-purpose FE assembly procedure were proposed in [11] and a complete edge-based compressible solver is described in [12]. To our best knowledge however, no GPU-based unstructured OpenCL FE solver has yet been published.

The implementation of a complete CFD solver on CPU requires a very large effort, not only because of the complexity in the solver itself but also because of all the surrounding issues, such as I/O, boundary conditions, and so on. An efficient GPU code is far harder to implement, considering complex architecture of modern GPUs. The development of the current work leverages the investment made over the last decade in developing the code Kratos [13] which provides all of the I/O capabilities as well as the necessary control mechanisms to allow the solution of complex engineering problems. Kratos is a free and open source framework for building multidisciplinary FE programs in an object-oriented fashion. More information about Kratos can be found in [13].

DOI: 10.1002/nme

Current work describes the redesign of an existing shared memory parallel CFD code implemented using OpenMP to fit in the framework provided by the OpenCL standard. The open-source package developed can be downloaded together with the Kratos distribution. Instruction for obtaining and building the code can be found on the Kratos Wiki page [14]. In order to harness the full computing power of multicore and many-core architectures, the FE computations are re-organized in an edge-based fashion, thus allowing the high level of parallelism needed to leverage the potential of modern hardware. A major amount of effort was spent over the years in the optimization of the CPU-based solver; consequently, the new OpenCL-based solver is up to a real-life test against a highly optimized CPU-based shared memory parallel solver. This is thus an important aspect of the current work. Such comparison is performed over a variety of hardware, benchmarking platforms of different ages so to give a clear view of the current status.

The layout of the paper is as follows. First, a brief description of the FE formulation is presented, providing details of the edge-based approach chosen. Some details of the CPU-based OpenMP solver are given in Section 3. The implementation and optimization of the solver in OpenCL platform is described in Section 4. Finally, benchmarks results and discussion are given in Section 5. Section 6 will conclude the paper.

2. EDGE-BASED FINITE ELEMENT FORMULATIONS

The current paper focuses on the solution of the convection-diffusion equation. The mathematical structure of the problem is described (assuming a constant density and conductivity) by

$$\frac{\partial \phi}{\partial t} + \mathrm{v} \bullet \nabla \phi - k \Delta \phi = Q \tag{1}$$

where ϕ is a scalar, v represents the convection velocity, k is a measure of the conductivity of the media, and Q is a source term. Despite the ease in including the diffusion term, the focus is mostly centered through the paper on the convection dominated case (and particularly on the pure-convection case, obtained by setting $k = 0$).

In order to proceed to the numerical solution, this equation has to be discretized both in space and time. Using FE shape functions and applying a standard Galerkin approach, the discretization results in the system

$$\underline{\underline{\mathbf{M}}} \frac{\partial \underline{\phi}}{\partial t} = \underline{\mathbf{Q}} - \left(\underline{\underline{\mathbf{C}}}\,(\underline{\mathbf{v}}) + k \underline{\underline{\mathbf{L}}} \right) \underline{\phi} \tag{2}$$

in which all the discrete contributions are represented using one or two underlines to represent vector or matrices. By introducing the nodal indices I and J, we can explicitly write the different contributions as

$$\underline{\underline{\mathbf{M}}}^c_{IJ} := \int_\Omega N_I N_J \, d\Omega \tag{3}$$

$$\underline{\underline{\mathbf{C}}}_{IJ}\,(\underline{\mathbf{v}}) := \int_\Omega N_I \underline{\mathbf{v}} \bullet \nabla N_J \, d\Omega \tag{4}$$

$$\underline{\underline{\mathbf{L}}}_{IJ} := \int_\Omega \nabla N_I \bullet \nabla N_J \, d\Omega \tag{5}$$

where for a given row I the only nonzero columns correspond to the indices J of the nodes that are first neighbors of the node I, that is, those that share one edge of the tetrahedrization with the node I. Although the consistent mass matrix $\underline{\underline{\mathbf{M}}}^c_{IJ}$ has important properties, it is not convenient in application with explicit time integration schemes. We thus introduce the lumped mass matrix $\underline{\underline{\mathbf{M}}}_{II} := \sum_J \underline{\underline{\mathbf{M}}}^c_{IJ}$ which we will use in all of the following developments.

DOI: 10.1002/nme

The discrete system in Equation (2) is known to lead to a spatially unstable solution for the convection dominated case. The problem can be solved using various stabilization techniques. The one we favor here can be derived using either FIC [15, 16] or OSS [17] approaches, to give an additional nonlinear term $\underline{\mathbf{S}}\left(\underline{\mathbf{v}}, \underline{\phi}\right)$ in the form

$$\underline{\mathbf{S}}_{IJ} := \tau_{II}\left(\int_\Omega \nabla N_I \bullet (\mathbf{v}\otimes\mathbf{v}) \bullet \nabla N_J d\Omega - \int_\Omega \nabla N_I \bullet (\mathbf{v}\otimes\mathbf{v}) \bullet N_J \underline{\mathbf{\Pi}}_J d\Omega\right) \tag{6}$$

where $\underline{\mathbf{\Pi}}$ represents the L^2 projection of the gradient of $\underline{\phi}$ onto the FE mesh

$$\underline{\mathbf{\Pi}}_I := \underline{\underline{\mathbf{M}}}^{-1} \int_\Omega N_I \nabla N_J \underline{\phi}_J d\Omega \tag{7}$$

and τ is a scalar, which we will take as

$$\tau_{II} := \left(\frac{\beta}{dt} + \frac{k}{h_I^2} + \frac{\|\underline{\mathbf{v}}\|}{h_I}\right)^{-1} \tag{8}$$

The new symbol h_I represents a characteristic length of the elements around the node I, which we can take for 3D cases, as $h_I := \left[\mathbf{M}_{II}^{1/3}\right]$. By introducing a residual operator $\underline{\mathbf{R}}$,

$$\underline{\mathbf{R}}_I := \underline{\mathbf{Q}} - \left(\underline{\underline{\mathbf{C}}}(\mathbf{v}) + k\underline{\underline{\mathbf{L}}}\right)\underline{\phi} - S\left(\underline{\mathbf{v}}, \underline{\phi}\right) \tag{9}$$

the problem can now be cast in the form

$$\underline{\underline{\mathbf{M}}}\frac{\partial \underline{\phi}}{\partial t} = \underline{\mathbf{R}}\left(\underline{\mathbf{v}}, \underline{\phi}\right) \tag{10}$$

which is suitable for being advanced in time using an explicit time integration scheme, such as the Runge–Kutta.

The resulting stabilized problem can be implemented in an FE code without any particular difficulty. The FE assembly needed however, although parallelizable, requires some degree of synchronization if standard techniques are used. In this work, we present a way to perform the FE assembly process in a way that allows exploiting the full parallel potential of modern GPUs. The main idea is to precompute all of the integrals that depend exclusively on the mesh geometry, recasting the operations so that the whole residual calculation can be computed on the basis of such precomputed integrals. It turns out that the computation of all the contributions described can be written in terms of only three operators to be precomputed for each edge IJ of a given FE mesh. A complete justification of the technique used falls beyond the scope of current work and can be found in [18]. In order to explain the basic concept, we can focus on the scalar Laplacian operator. The main idea is to take advantage of the partition of unity property of the FE, which guarantees that, at any point in space, the shape functions sum to 1 and their derivative sum to 0, that is $N_I = 1 - \sum_{I\neq J} N_J$ and $\nabla N_I = -\sum_{I\neq J} \nabla N_J$. Such property implies that the operation

$$\underline{\mathbf{r}}_I = \sum_J \int_\Omega \nabla N_I \nabla N_J \underline{\phi}_J = \int_\Omega \nabla N_I \nabla N_I \underline{\phi}_I + \sum_{I\neq J} \int_\Omega \nabla N_I \nabla N_J \underline{\phi}_J \tag{11}$$

can be rewritten as

$$\underline{\mathbf{r}}_I = -\sum_{I\neq J} \int_\Omega \nabla N_I \nabla N_J \underline{\phi}_J + \sum_{I\neq J} \int_\Omega \nabla N_I \nabla N_J \underline{\phi}_J \tag{12}$$

or in a more concise form as

$$\underline{\mathbf{r}}_I = \sum_{I\neq J} \int_\Omega \nabla N_I \nabla N_J \left(\underline{\phi}_J - \underline{\phi}_I\right) = \sum_{I\neq J} \underline{\underline{\mathbf{L}}}_{IJ}\left(\underline{\phi}_J - \underline{\phi}_I\right) \tag{13}$$

DOI: 10.1002/nme

which is the technique we use in the following. For the Laplacian term described, the rewriting is exact as long as no variable-diffusion term is needed. Some approximations are however needed in the case of variable diffusivity as well as for the description of the convective term [19]. The approach followed does not coincide exactly with the edge-based approaches [20] nor with nodal based techniques, but rather makes an attempt to blend the characteristics of the two methods. We try to preserve the stabilization used in Codina's work [19] and also taking advantage of the intrinsic conservation properties of Soto's proposal [20].

In our approach, the integrals to be precomputed are the following (to be stored for each pair of indices $I \neq J$):

$$\underline{\underline{\mathbf{L}}}^{d}_{IJ} := \int_{\Omega} \nabla N_I \otimes \nabla N_J \, d\Omega \tag{14}$$

$$\underline{\nabla}_{IJ} := \int_{\Omega} N_I \nabla N_J \, d\Omega \tag{15}$$

$$\underline{\mathbf{G}}_{IJ} := \int_{\Omega} \nabla N_I N_J \, d\Omega \tag{16}$$

The coefficients of the *standard* Laplacian operator, which are needed to write some of the operations, can be recovered as

$$L_{IJ} := Tr\left(\underline{\underline{\mathbf{L}}}^{d}_{IJ}\right) = \sum_{k=1}^{3} \left(\underline{\underline{\mathbf{L}}}^{d}_{IJ}\right)_{kk} \tag{17}$$

On the basis of such operators and using the conservative integration rule described in [19], we obtain approximate integration rules for the fast computation of the residuals. For example, in the case of convective contribution, we have

$$\underline{\mathbf{A}}_I := \sum_J \underline{\underline{\mathbf{C}}}_{IJ} \phi_j \approx \sum_{I \neq J} \left((\underline{\nabla}_{IJ}) \bullet \underline{\mathbf{v}}_I\right)\left(\underline{\phi}_J - \underline{\phi}_I\right) \tag{18}$$

and in case of diffusive contribution

$$\underline{\mathbf{B}}_I := k \sum_I \underline{\underline{\mathbf{L}}}_{IJ} \phi_j = \sum_{I \neq J} \underline{\underline{\mathbf{k}}}_{IJ} \left(Tr\left(\underline{\underline{\mathbf{L}}}^{d}_{IJ}\right)\right)\left(\underline{\phi}_J - \underline{\phi}_I\right) \tag{19}$$

The stabilization contribution is divided in two parts: a *low order* one defined as

$$\underline{\mathbf{S}}^{low}_I := \sum_{I \neq J} \left(\sum_{k=1}^{3} \sum_{l=1}^{3} (\underline{\mathbf{v}}_I)_k (\underline{\mathbf{v}}_I)_l \left(\underline{\underline{\mathbf{L}}}^{d}_{IJ}\right)_{kl} \right) \left(\underline{\phi}_J - \underline{\phi}_I\right) \tag{20}$$

and a *high order* part that can be written as

$$\underline{\mathbf{S}}^{high}_I := \sum_{I \neq J} \left((\underline{\nabla}_{IJ}) \bullet \underline{\mathbf{v}}_I\right)\left(\underline{\mathbf{v}}_J \bullet \underline{\mathbf{\Pi}}_J - \underline{\mathbf{v}}_I \bullet \underline{\mathbf{\Pi}}_I\right) \tag{21}$$

To allow the computation of $\underline{\mathbf{S}}^{high}_I$, the projection terms $\underline{\mathbf{\Pi}}$ have to be precomputed, which can be achieved by performing

$$\underline{\mathbf{\Pi}}_I := \underline{\underline{\mathbf{M}}}^{-1}_{II} \sum_{I \neq J} (\underline{\nabla}_{IJ}) \left(\underline{\phi}_J - \underline{\phi}_I\right) \tag{22}$$

Finally, the residual can be computed by summing the different contributions as

DOI: 10.1002/nme

$$\underline{\mathbf{R}}_I\left(\underline{\mathbf{v}},\underline{\phi}\right) = \underline{\mathbf{A}}_I + \underline{\mathbf{B}}_I + \left(\underline{\mathbf{S}}_I^{low} - \underline{\mathbf{S}}_I^{high}\right) \tag{23}$$

The computation technique described represents an approximation of the original FE approach. The advantage gained is that the computation of the residuals is now purely node-based and as a consequence perfectly parallel, in the sense that each index I can be computed independently.

In order to apply the Runge–Kutta scheme to advance in time the solution, we will now assume given the value $\underline{\phi}_n$, which represents the known value of the solution at the time step n and the convection velocities both at the beginning $_n$ and at the end of the time step $_{n+1}$. Under such assumption, the algorithm proceeds as follows:

1. $\underline{\mathbf{v}} = \underline{\mathbf{v}}_n \quad \underline{\phi} = \underline{\phi}_n$
2. $\underline{\mathbf{r}} = \underline{\underline{\mathbf{M}}}^{-1}\underline{\mathbf{R}}\left(\underline{\mathbf{v}},\underline{\phi}\right)$
3. $\underline{\mathbf{w}} = \frac{\Delta t}{6}\underline{\mathbf{r}} \quad \underline{\phi} = \underline{\phi}_n + \frac{\Delta t}{2}\underline{\mathbf{r}}$
4. $\underline{\mathbf{v}} = \frac{1}{2}\underline{\mathbf{v}}_n + \frac{1}{2}\underline{\mathbf{v}}_{n+1}$
5. $\underline{\mathbf{r}} = \underline{\underline{\mathbf{M}}}^{-1}\underline{\mathbf{R}}\left(\underline{\mathbf{v}},\underline{\phi}\right)$
6. $\underline{\mathbf{w}} = \underline{\mathbf{w}} + \frac{\Delta t}{3}\underline{\mathbf{r}} \quad \underline{\phi} = \underline{\phi}_n + \frac{\Delta t}{2}\underline{\mathbf{r}}$
7. $\underline{\mathbf{v}} = \frac{1}{2}\underline{\mathbf{v}}_n + \frac{1}{2}\underline{\mathbf{v}}_{n+1}$
8. $\underline{\mathbf{r}} = \underline{\underline{\mathbf{M}}}^{-1}\underline{\mathbf{R}}\left(\underline{\mathbf{v}},\underline{\phi}\right)$
9. $\underline{\mathbf{w}} = \underline{\mathbf{w}} + \frac{\Delta t}{3}\underline{\mathbf{r}} \quad \underline{\phi} = \underline{\phi}_n + \Delta t\underline{\mathbf{r}}$
10. $\underline{\mathbf{v}} = \underline{\mathbf{v}}_{n+1}$
11. $\underline{\mathbf{r}} = \underline{\underline{\mathbf{M}}}^{-1}\underline{\mathbf{R}}\left(\underline{\mathbf{v}},\underline{\phi}\right)$
12. $\underline{\phi} = \underline{\mathbf{w}} + \frac{\Delta t}{6}\underline{\mathbf{r}}$

The selected time scheme is explicit which implies intrinsically a restriction in the time step.

3. OPENMP IMPLEMENTATION

The implementation of an edge-based data structure can be performed efficiently both with and without using object-oriented approach. In our proposal, we favor the second choice, defining an implementation that encapsulates all of the edge operations within a class *edge*. Each edge is designed as a container class which holds two arrays of size 3, that is $\underline{\nabla}_{IJ}$ and $\underline{\mathbf{G}}_{IJ}$, a 3 by 3 matrix $\underline{\mathbf{L}}_{IJ}^{d}$ as well as the term IJ of the consistent mass matrix, for a total of exactly 16 `doubles`. All of the functions operating over edges are then defined as member methods of the class edge. A complete list of the functions is provided in Table I.

In order to efficiently perform all of the computations, only the edges that effectively exist in the FE discretization are required. The edges that correspond to the diagonal terms are not needed and consequently are not stored. Observing the structure of the calculation algorithms, it is easily seen that the computation of the residuals (and of the projections) resembles closely that of sparse matrix vector multiply (SpMV), in the sense that two main loops are involved: an outer one of the node I and then a secondary one on the J's that correspond to the neighbors of I. The idea we followed in our implementation is to use a compressed sparse row (CSR) data structure to store the edges and to mimic the behavior of SpMV algorithms in writing the edge contributions to the RHS. The advantage of our object-oriented approach is that the edge concentrates most of the data needed for the computations, which vastly improves the data locality and consequently the cache efficiency. In order to facilitate the implementation of different algorithms, the edge structure predefines auxiliary functions to assemble the edge contribution. This implies that for example, a function `Add_gradp()` is defined to assemble to the residual (RHS) the gradient contribution corresponding to the edge of interest. On the basis of such approach, the function to compute the residual,

DOI: 10.1002/nme

Table I. Functions operating over edges.

Function	Expression	Result
{Add,Sub}_Gp	$\mathbf{r}_I = \mathbf{r}_I \pm \int_\Omega \nabla N_I N_J \underline{\phi}_J \Omega$	Vector
{Add,Sub}_gradp	$\mathbf{r}_I = \mathbf{r}_I \pm \int_\Omega N_I \left(\nabla N_J \underline{\phi}_J\right) \Omega$	Vector
{Add,Sub}_D_v	$\mathbf{r}_I = \mathbf{r}_I \pm \int_\Omega N_I \left(\nabla N_J \bullet \underline{\mathbf{v}}_J\right) \Omega$	Scalar
{Add,Sub}_div_v	$\mathbf{r}_I = \mathbf{r}_I \pm \int_\Omega \nabla N_I \bullet \left(N_J \underline{\mathbf{v}}_J\right) \Omega$	Scalar
ScalarLaplacian	$Tr\left(L^d_{IJ}\right)$	Scalar
{Add,Sub}_ConvContrib	$\mathbf{r}_I = \mathbf{r}_I \pm \int_\Omega N_I \left(\nabla N_J \bullet \underline{\mathbf{v}}_J\right) \underline{\phi}_J \Omega$	Scalar
ConvStab_LOW	$\int_\Omega \nabla N_I \bullet (\mathbf{v} \otimes \mathbf{v}) \bullet \nabla N_J d\Omega$	Scalar
ConvStab_HIGH	$\int_\Omega \nabla N_I \bullet (\mathbf{v} \otimes \mathbf{v}) \bullet \left(N_J \underline{\mathbf{\Pi}}_J\right) d\Omega$	Scalar
{Add,Sub}_ViscContrib	$\mathbf{r}_I = \mathbf{r}_I \pm \int_\Omega k \nabla N_I \nabla N_J \underline{\phi}_J \Omega$	Scalar

in pseudocode, has the following form (note that the projections are assumed to be calculated in another, similar, loop)

```
(parallel) loop I on all nodes

 // Obtain data for node I
 v_I = Gather(velocity, I)
 Pi_I = Gather(Pi, I)
 phi_I = Gather(phi, I)
 tau_I = Gather(tau, I)

 rhs_I = 0.00
 temp_I = 0.00

 loop J in neighbors of I
  edge_IJ = GatherEdge(I, J)

  v_J  = Gather(velocity, J)
  Pi_J = Gather(Pi,J)
  phi_J = Gather(phi, J)

  rhs_I -= edge_IJ.ConvectiveContrib(v_I, phi_i, v_J, phi_J)

  temp_I -= edge_IJ.StabLowOrder(v_I, phi_I, v_J, phi_J)
  temp_I += edge_IJ.StabHighOrder(v_I, Pi_I, v_J, Pi_J)

// Write the results to the database
rhs_I += tau_I * temp_I
```

As shown in the pseudocode, the different functions all require the edge data, and thus can be efficiently implemented in C++ as members of the class edge. For each value of the counter I, the algorithm gathers the information needed from all of the neighboring nodes; a shared access is therefore needed to a number of different arrays. The only writing access is to the variable `rhs_I`, which implies that the procedure can be easily parallelized in OpenMP by splitting the main loop over the nodes. The use of a CSR data structure in storing the neighbor relationships allows an optimal parallel access to the variables involved. Furthermore, encapsulating the data inside the edge

DOI: 10.1002/nme

data structure allows all of the edge data (16 `doubles`) to be fetched at once with a single indirect access to the edge class.

In our implementation, two distinct databases are used. First, the classic node oriented Kratos data structure [13], which is used for data I/O, and a second array-based one, specifically created to provide fast access to the variables needed. The number of copies between the different databases was minimized, so to reduce the impact of the copy time (impact is estimated to be around 10%–15% of the overall computation time). All of the copies in both directions were also parallelized. Such parallelization was found to be surprisingly effective on multicore systems.

4. OPENCL IMPLEMENTATION AND OPTIMIZATION

Over the years, many scientific codes and libraries have been developed. The emerging platform of modern GPUs has received a lot of attention from scientists and developers, and many of them are considering the migration of their code to this platform. Porting the available codes will help reduce the effort needed to make these codes benefit from such new hardware platform. Here, we describe the process of porting the aforementioned OpenMP solver to OpenCL platform. We will then discuss various stages of optimization of the code. Finally, we will point out some issues which might be of importance.

4.1. Porting process

The difficulty in porting existing CPU-oriented codes to GPUs is intrinsically related to the programming model used. Although on CPUs, a single addressing space is used implying that the user does not need to take care explicitly of the memory placement; OpenCL relies on a heterogeneous programming model in which the code is divided between a *host* and the computing *device*s. The management of the overall process is left to the host side, which assumes explicitly the control of the calculations to be launched on the devices, deciding which *kernel*s should be executed on the device and which portions of memory need to be transferred. In the current context, the CPU will be used as a host, and a single GPU will be considered as computing device. For the specific problem of interest, all of the computational steps to be performed were moved to the device side, whereas the overall initialization process is performed on the host side. That is, the host prepares all of the data structures to be used in the computations and transfers them to the device one single time at the beginning of the calculations. After such initial step, all the computations are performed on the device. Host and device only synchronize the data needed for I/O.

As previously mentioned, the original OpenMP code was written in C++ using an object-oriented programming style. Despite the existence of bindings for many different languages like C++, the OpenCL language itself is restricted to the C99 standard in writing the device code. The main difficulty in the porting process was thus related to the definition of an equivalent storage for the edge data that had to be written within the restrictions of the OpenCL standard. Two alternatives were considered, namely the use of a `struct` to store the different fields needed and the use of a the native data type `double16`. Although at first, the former appeared to be attractive to allow automatic vectorization of operations, the latter was finally chosen as it was found to allow a better performance. The class member functions were then emulated by the introduction of helper functions accepting the edge data type as input data. Given the unavailability of C++ templates, support for 2D cases was dropped, and the code was specialized to the 3D case. A helper C++ library, `opencl_interface.h`, was developed for easy interfacing with OpenCL. This library encapsulates the most important parts of OpenCL functionality in an object-oriented manner and is designed to allow future support for multidevice programs.

The solution steps and in particular, the `Solve()` function was analyzed and divided into independent chunks. These chunks were transformed into OpenCL kernels with code pieces inside their outermost loop as *work item*s. This can be seen in the following pseudocode:

DOI: 10.1002/nme

```
// OpenMP code

void Solve()
{
 // Initializations here

 ...
 // Parallel loop #1

 parallel loop I on all nodes

  ...

 // Parallel loop #2

 parallel loop I on all nodes

  ...

 // Parallel loop #3

 parallel loop I on all nodes

  ...
}
```

```
// OpenCL code

void Solve()
{
  // Initializations here

  ...
  // Set arguments and launch
  // OpenCL kernel Solve1()

  ...
  ...

  // Set arguments and launch
  // OpenCL kernel Solve2()

  ...
  ...

  // Set arguments and launch
  // OpenCL kernel Solve3()

  ...
  ...
}
```


The next step of the porting process was to rewrite the methods of edge class in the form of OpenCL functions. In the first place, the existing loops were unrolled by hand. No vectorization was performed at this stage as we tried to keep the original form of the code intact as much as possible to avoid introducing bugs. Several changes were needed because of the differences between the OpenCL C99 language and the C++ language. Down-casting and preprocessor macros played an important role, making the code much more human readable. The OpenCL device codes were placed inside `.cl` files for easy maintenance and editing. The aforementioned OpenCL interface library eased loading and executing of OpenCL kernels from these files. In the next step, OpenCL buffers were declared and necessary support code as well as calls to kernels were provided. At this stage, primary testing was performed and the correctness of the results was verified.

4.2. *Optimization process*

The optimization of the code can be seen from very different aspects. We tried several options to find out the bottlenecks and potential areas in which the code can be further improved. We classified the process in several stages from easy-to-adopt actions to those requiring several changes in the code. After each step, an automated benchmarking system verified the correctness of the results as well as increase or decrease in the execution time.

Vectorization of loops over the components of three vectors. The first stage of optimization was to take advantage of OpenCL native vector data types inside all of the suitable methods of the edge class. Apart from the performance impact, such approach allowed reducing the size of the code considerably, consequently minimizing the risk of programming errors. On the other hand, the use of vectorized operations allows using ATI hardware in an efficient manner, achieving a higher ALU vector packing efficiency. To achieve the required portability over different GPUs from all major hardware vendors, we needed to support both OpenCL 1.0 and 1.1 standards. OpenCL 1.0 does not support `double3` type, which is frequently needed in the code, and version 3.2 of NVIDIA CUDA Software Development Kit (SDK) used for writing this paper, does not support OpenCL

DOI: 10.1002/nme

1.1. To work around this problem, we defined some preprocessor macros which map to appropriate function based on the standard supported by the SDK.

Using OpenCL library functions. In the next stage, we attempted taking advantage of OpenCL library functions instead of custom written code. Relevant examples are the functions `mix()` and `dot()`. This stage of optimization allowed reducing the code base, leading to a cleaner and more efficient implementation, factory tuned to benefit from future advances in software and hardware.

Native functions. OpenCL introduces a set of *native* functions which has an implementation defined accuracy and input range. These functions typically deliver better performance and are favored when accuracy is not as important as speed. Unfortunately, at the moment of writing, native functions are not yet completely supported, as either the `double` versions are not provided, or they are only supported on CPU device (or not yet reliably supported between different SDKs). As a temporary solution while waiting for the SDKs to mature, preprocessor macros were used to allow the use of native versions when available. We should remark that the problem of interest is bandwidth limited rather than compute limited. As a consequence, the use of fast native functions has only a minor impact on the overall timings.

Simple kernel optimizations. We tried performing some simple optimizations, such as factoring out common parts of expressions. Compilers might be able to do some of these optimizations. Of course, hand tuning had to be attempted before relying on the compiler.

Using a custom `struct` for storing edge data. As previously described, we had chosen a `double16` for storing all required fields. It seemed logical at that time, that using a `struct` of native OpenCL vector data types such as `double3` could boost the performance by benefiting from vector load and store capabilities of the GPUs. After a few experiments, it proved that the reasoning was not quite true here, and the performance was degraded. The reason could be described in the higher amount of global memory which needed to be fetched because of alignment restriction of OpenCL `structs`. So, we decided to revert back to the original idea of using `double16`'s.

Vectorization of the algorithm. This was one of the most appealing ideas, when the similarities of code to SpMV kernel was considered. The main idea is to use several threads per each CSR row, like what is described in [21]. But as the average number of edges per node (i.e. the number of nonzeros in each row) was rather low, no performance boost would be expected, as the previous experiences on SpMV kernels had shown.

Following general optimization guidelines for GPUs. We tried some simple guidelines for optimization of device code for GPUs, such as using *ternary* operator, instead of some `if`'s. This kind of optimization might benefit cases in which multiply-and-add operations can replace some kinds of `if`'s.

Reducing global memory usage. All of the aforementioned experiences proved that the code is highly memory bound, and we need to lower global memory usage. We tried caching some of the values read from global memory in local variables to avoid accessing them each time in the loops. Sometimes this kind of optimization seemed to degrade the performance. After some investigation using GPU profilers, it appeared that this caused high register usage in some cases. Considering limited number of general purpose registers in the GPU register file, high register usage decreases the number of simultaneous threads that can be run. This will decrease the performance, as the GPUs tend to hide global memory latency by dividing the threads into several wavefronts and rescheduling a new wavefront when the current one is waiting for a global memory request to be completed. So, we started to minimize the register usage, which is not an easy task in most of the cases. One needs a very good knowledge of the hardware of modern GPUs to accomplish this task successfully. For example, AMD OpenCL Programmers' Guide [22] states that AMD GPUs can use the result of last operation without using extra registers or register file bandwidth. Hence, we re-ordered some of

DOI: 10.1002/nme

the independent calculations in kernels to use this facility. This kind of optimization decreased the number of registers used and helped the overall performance.

Using local data storage. Trying to maximize the number of simultaneous threads executed dictates a very small local data storage per thread. Also, this kind of storage must be used for variables that can be shared among the threads. Recalling the similarities of the algorithm used to SpMV, it is clear that the lower bound of the execution range for one thread is the upper bound for another. Thus, for N threads, we need $N + 1$ global memory reads to have all upper and lower bounds, and appropriate values can be shared among threads in a *work group*. This was implemented in the code to reduce the global memory access.

4.3. Other issues and considerations

After performing the aforementioned stages of optimization, performance of the code seemed to be comparable with the optimized OpenMP code, even on modern CPUs available today. However, still, there were some limitations in using the code; for example, the size of the problem which could be solved was limited. This prevented running large problems, where GPUs could perform even better. In OpenCL, the amount of global memory which can be allocated at once is limited. The host code can query this limit using `clGetDeviceInfo()` function. This value is very different among the available SDKs. For example, for AMD Stream SDK 2.2, this value is 128 MB. This is not a huge value when large problems are considered. One of the workarounds could be using OpenCL *image*s. Images are less limited in this sense and also benefit from a caching mechanism but tend to be much more complex in nature. We chose to use images to store edge data, which consumes the most of used storage. The first challenge here is that images are not designed to be treated as general buffer objects. Reading and writing images can only be carried out using OpenCL-provided functions. Also, the data has to be provided in the form of 2D or 3D images, with rectangular shape, limited in dimension. This implies some extra work on the host side as well as on the device.

Another issue which makes using images more difficult is the necessity of accessing the image data in terms of `float4`'s. This means that we have currently no way of reading `double`'s directly from images. As each `double` occupies 64 bits, which is twice a `float`, we tried to convert the `float4`'s to `double2`'s. Unfortunately, we found no working, portable way to do this. For example, the `as_double2()` function was found to be broken in NVIDIA's SDK. Another workaround, using `union`'s to cast the types introduced by OpenCL simply did not compile on some platforms. Finally, we came up with a solution that was dependent on the endianness of the host and device. It consisted of casting `float4`'s to `uint4`'s using bitwise operations to combine them and finally casting them into `double2`'s.

After getting all of the pieces to work correctly, it appeared that the performance degraded slightly. It was expected, as there is some extra overhead for creating an image and also there would not be any benefit from the caching mechanism. The clear advantage was however that much larger problems could be run, and better speedups would be obtained.

5. BENCHMARKS

In this section, we analyze the performance of the OpenCL implementation versus the optimized OpenMP solver described in Section 3. We attempt to study the effect of porting the code to GPU platform, as well as relative performance of different hardwares over the time. For a comprehensive study of the code behavior, a range of platforms with different relative age from various vendors were chosen. The specifications of these platforms can be found in Table II. All platforms use 64 bit Ubuntu operating system, with the version given. The default system compiler (GCC) was always used. Platforms 1–4 use NVIDIA CUDA SDK 3.2, whereas platform 5 uses AMD Stream SDK 2.3, all with the latest driver available.

It is noteworthy that although a given instance of the convection stabilization is used in our implementation, other alternatives could be easily implemented on the top of the existing infrastructure,

DOI: 10.1002/nme

Table II. Platforms description.

Platform	CPU	GPU	OS / GCC version
1	Intel Core 2 Duo E6600	NVIDIA GTX 260	10.04 / 4.4
2	AMD Phenom Quad core 9950	NVIDIA GTX 280	10.10 / 4.4
3	Intel Core 2 Quad Q9550	NVIDIA GTX 280	10.04 / 4.4
4	Intel Core i7 920	NVIDIA GTX 285	9.04 / 4.3
5	Intel Core i7 920	ATI Radeon HD 5870	10.04 / 4.4

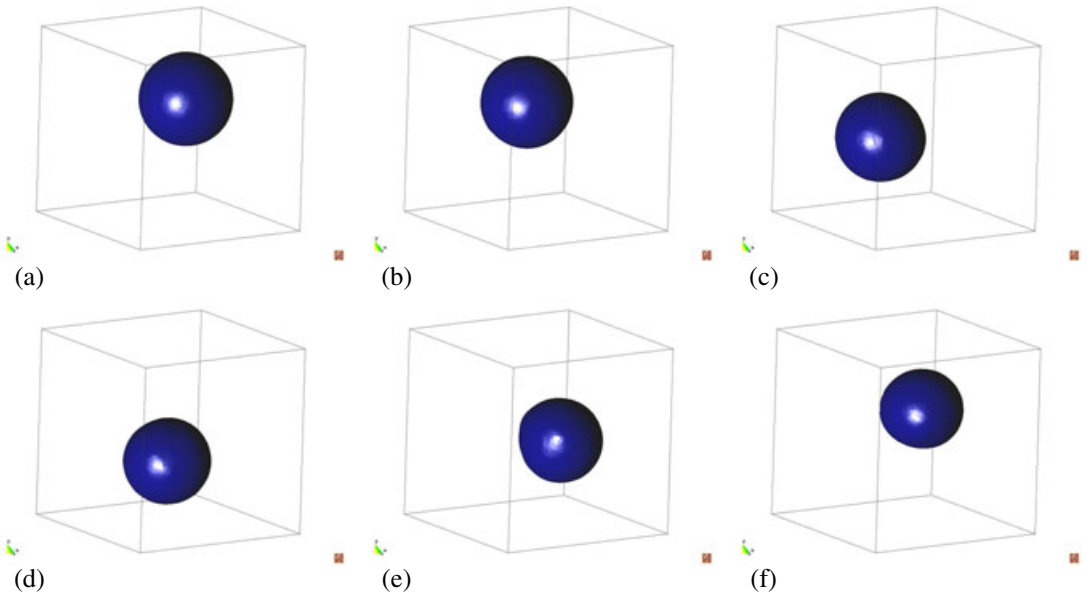

Figure 1. Evolution of the iso-surface of a distance function subjected to convection.

leading only to marginal differences in the overall computational effort. Taking in account such objective, the benchmarks are intended to discuss the *computational efficiency* of the method rather than concentrating on the characteristics of the formulation.

5.1. 'Gaussian hill' convection

The first test case selected for our studies is the 3D equivalent of the classic 'Gaussian hill' pure convection [23], a test traditionally used in evaluating the quality of existing convection diffusion algorithms. The reason of our choice is, however, the ease in reproducing the test setup rather than the specific feature of the test itself.

The problem consists of a cube of side 1 m centered at the origin. A rotational convection velocity of the type $v_x := -y\,;\ v_y := x; v_z := 0$ is prescribed and a 'bubble' centered at $(1/8\,m, 1/8\,m, 0\,m)$ is provided in the beginning of the test. The exercise consists of convecting (with the minimum possible level of deformation) of such bubble for one or more complete turns. In order to simplify the numerical experiment, three different levels of refinement are considered by using nested grids. No renumbering is performed, implying that the locality of the data is somewhat worse when finer meshes are used. A view of some snapshots can be found in Figure 1.

Table III shows the timings on different hardware configuration and for different refinement levels. Setup times are not taken in account, in any case they appear to be sensibly minor (less than 5%) compared with the computational times presented here. The systems 4 and 5 feature the same CPU; nevertheless, system 4 is equipped with an older OS (Ubuntu 9.04, GCC 4.3), whereas the test system number 5 uses a more modern operative system and compiler (Ubuntu 10.04, GCC 4.4).

DOI: 10.1002/nme

Table III. Results for 'Gaussian hill' convection test case.

Platform	0-level refinement			1-level refinement			2-level refinement		
	OpenMP	OpenCL	Ratio	OpenMP	OpenCL	Ratio	OpenMP	OpenCL	Ratio
1	4.633 s	2.153 s	2.2	77.587 s	22.320 s	3.5	1280.808 s	298.245 s	4.3
2	1.277 s	1.590 s	0.8	22.854 s	15.311 s	1.5	384.770 s	202.578 s	1.9
3	0.686 s	1.987 s	0.3	28.878 s	15.312 s	1.9	490.130 s	184.786 s	2.7
4	1.000 s	1.600 s	0.6	14.820 s	13.110 s	1.1	236.820 s	167.320 s	1.4
5	0.646 s	1.043 s	0.6	13.330 s	8.790 s	1.5	212.269 s	111.895 s	1.9

Table IV. Results for complex domain test case.

Performance counter	Value
ALUBusy	9.01
ALUFetchRatio	8.63
ALUPacking	77.12
CacheHit	23.97
FetchUnitBusy	84.32
PathUtilization	100.0%
LDSBankConflict	0.0%

Such difference is reflected in the OpenMP performance, which appears to be slightly worse on the older system.

The analysis of the profiling reported in Table IV (on the test setup number 5) suggests that the computations are bandwidth limited. Here, we report some of the performance counters for the most time consuming kernel, `CalculateRHS()`. The results of the other kernels (in particular the computation of the projections) are in line with this results. Profiling was executed for 1-level of refinement.

Interestingly, the profiling also suggests that the number of computations per memory access is relatively low (ALUFetchRatio value, [22]). This situation may result in a more favorable speedup for more complex kernels, such as the ones that will be needed in dealing with the Navier–Stokes equations.

5.2. Convection of a distance function over a complex domain

The second example considered was the convection of a scalar function over a complex domain. The geometric setup chosen was the so called 'Ahmed's body', a well-known test in CFD. The simulation of the fluid flow around the body was performed first using a different module of the Kratos. Slip condition was applied on all of the walls of the model. The resulting velocity field was written to a file which was then used as initial condition in benchmarking our convection solver. These velocities as well as the problem geometry is visualized in Figure 2. The evolution of an iso-surface of the function that is being convected is shown in Figure 3 at different instants in time. In this case, pure convection is considered.

The most interesting feature of the example is showing how the OpenCL solver can easily handle arbitrarily complex geometries, featuring in particular curved surfaces as well as details of different sizes, see Figure 4 for a view of the surface mesh used. This shows clearly the advantage of unstructured methods over alternative approaches based on the usage of regular grids. The example was run on a fairly uniform mesh composed of 1,881,977 elements and 338,382 nodes, which implied the usage of a very large percentage of the 1 GB available GPU memory.

The timings on the aforementioned platforms are reported in Table V. As expected, the OpenCL solver consistently outperforms the OpenMP version. It is also very interesting that the Core i7 CPU gains a factor of 6 over the oldest machine, probably because of the much higher memory bandwidth and to a lesser extent, to the higher thread count (4 vs. 2 in our test). The memory bandwidth was measured on all of the different systems using the 'STREAM' benchmark [24]. The benchmarking

DOI: 10.1002/nme

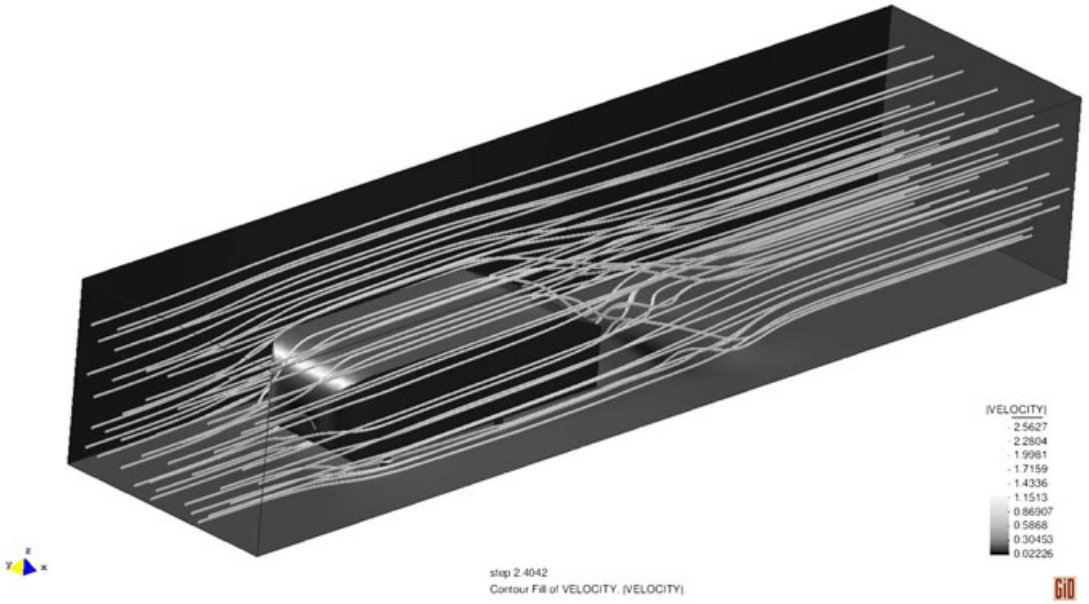

Figure 2. Velocity field view.

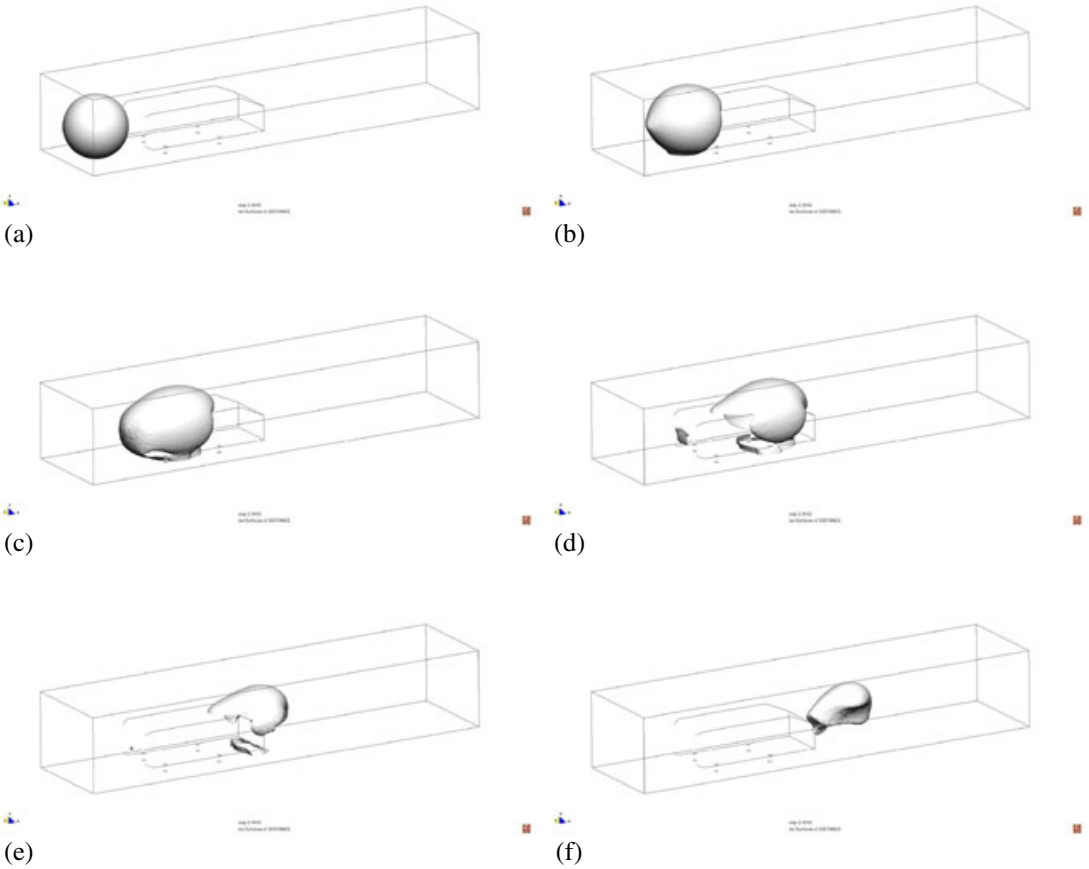

Figure 3. Evolution of the iso-surface of a distance function subjected to convection.

DOI: 10.1002/nme

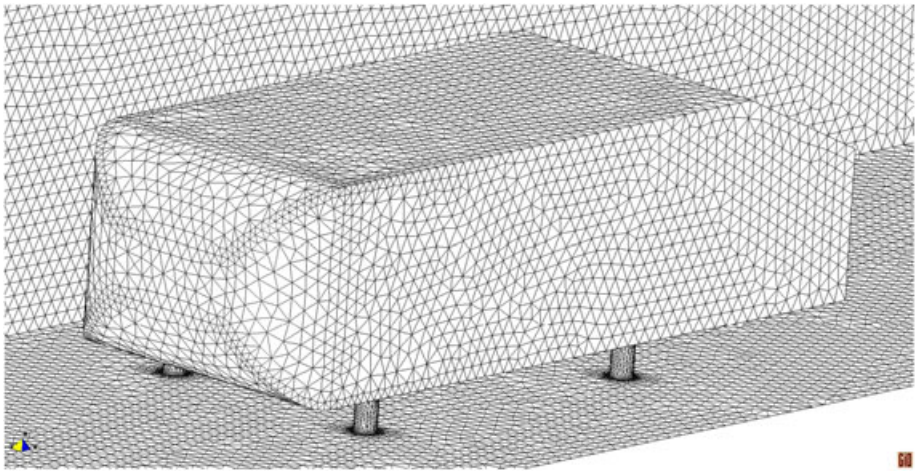

Figure 4. Detail of surface mesh.

Table V. Results for complex domain test case.

Platform	OpenMP	OpenCL	Ratio
1	17742.297 s	3819.723 s	4.6
2	5183.223 s	2619.693 s	2.0
3	6562.798 s	2761.287 s	2.4
4	3296.927 s	2328.521 s	1.4
5	2909.765 s	1656.754 s	1.8

Table VI. Memory bandwidth for CPU and graphical processing unit on the benchmark platforms.

Platform	RAM bandwidth (GB/s)		
	CPU (Add)	GPU (Add)	GPU (Copy)
1	2.11	61.86	95.14
2	7.62	71.16	118.16
3	5.03	71.44	119.49
4	13.50	80.77	130.95
5	16.53	80.62	117.60

software was compiled on all of the different platforms using maximal optimization and including OpenMP. It was observed that the OpenMP parallelization was important in achieving the maximum bandwidth for the test considered. The first column of Table VI reports the experimental system's bandwidth of the different platforms considered in our benchmark. The values reported are measurements of the 'Add' test case in which two vectors of `doubles` are added and are therefore a reliable performance indicator of the different systems. On the other hand, the second and third columns list the measured bandwidth of GPU memory on the benchmark systems. For a fair comparison of the memory bandwidth between CPU and GPU, we implemented two test cases on the GPUs. The first one, listed in the second column of the Table VI, is similar to the 'Add' test case on the CPU. The second test case, however, tries to find the maximum memory bandwidth of the GPU, by moving contents of a GPU buffer to another, using provided OpenCL functions. This is similar to the way the device-to-device bandwidth is measured in the NVIDIA SDK sample, in OpenCL way. The values reported should not be confused by host-to-device transfer bandwidth, which is limited by data transfer speed of PCIe bus. It can be observed how the performance gain is directly related to the relative improvement in the system's memory bandwidth.

DOI: 10.1002/nme

The OpenCL-based implementation allows the code to fall back on CPU when a GPU is not present. Nevertheless since the objective of the paper was to evaluate the GPU versus CPU performance, only little effort was devoted to optimizing the CPU-OpenCL version which would require, as a very minimum, tuning the appropriate number of threads to efficiently use the hardware. We thus preferred to refer to the highly optimized OpenMP version which was carefully tuned over the years for such benchmark comparison.

6. CONCLUSIONS

The developements presented in the current work show how a portable CFD framework can be effectively implemented in OpenCL. The last generations of GPU hardware appear to consistently outperform CPU hardware of the same generation, even when benchmarked against optimized parallel CPU-oriented implementations. Nevertheless, for the unstructured FE calculations we considered the performance advantage of GPUs vs. CPUs, althought noticeable (around twice faster), is not overwhelming. This result is in line with the findings in [25] which present a speedup of the order of 2.5 for a SpMV kernel against the corresponding parallel CPU implementation on the Core i7 CPU. This is mostly related to the non-regular memory acces pattern and to the relatively low arithmatic operation count per memory access.

ACKNOWLEDGEMENTS

The first author would like to thank the International Center for Numerical Methods in Engineering for kindly supporting this work. The work is also supported under the auspices of the 'eDams' project of the MEC (BIA2010-21350-C03-00) and of the ERC Advanced Grant 'RealTime' project AdG_2009325. The authors would like to acknowledge the help of Jordi Cotela Dalmau in performing the benchmarks.

REFERENCES

1. Lindholm E, Kilgard MJ, Moreton H. A user-programmable vertex engine. In *SIGGRAPH 01: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. ACM: New York, NY, USA, 2001; 149–158.
2. Purcell TJ, Buck I, Mark WR, Hanrahan P. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics* 2002; **21**(3):703–712.
3. Owens JD, Luebke D, Govindaraju N, Harris M, Krger J, Lefohn AE, Purcell TJ. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum* 2007; **26**(1):80–113.
4. NVIDIA CUDA. Available from: http://www.nvidia.com/object/cuda_home_new.html [15 December 2010].
5. Idelsohn SR, Oñate E, Del Pin F. The particle finite element method: a powerful tool to solve incompressible flows with free-surfaces and breaking waves. *International Journal for Numerical Methods in Engineering* 2004; **61**(7):964–984.
6. Larese A, Rossi R, Oñate E, Idelsohn S. Validation of the particle finite element method (PFEM) for simulation of free surface flows. *Engineering Computations* 2008; **25**(4):385–425.
7. Oñate E, Idelsohn S, Celigueta MA, Rossi R. Advances in the particle finite element method for the analysis of fluid-multibody interaction and bed erosion in free surface flows. *Computer Methods in Applied Mechanics and Engineering* 2008; **197**(19–20):1777–1800.
8. Ryzhakov P, Rossi R, Idelsohn S, Oñate E. A monolithic Lagrangian approach for fluid-structure interaction problems. *Computational Mechanics* 2010; **46**(6):883–899.
9. Rossi R, Oñate E. Analysis of some partitioned algorithms for fluid-structure interaction. *Engineering Computations* 2010; **27**(1):20–56.
10. Klockner A, Warburton T, Bridge J, Hesthaven JS. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics* 2009; **228**(21):7863–7882.
11. Cecka C, Lew AJ, Darve E. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering* 2010; **85**(5):640–669.
12. Corrigan A, Camelli F, Löhner R, Wallin J. Running unstructured grid-based CFD solvers on modern graphics hardware. *International Journal for Numerical Methods in Fluids* 2011; **66**(2):221–229.
13. Dadvand P, Rossi R, Oñate E. An object-oriented environment for developing finite element codes for multi-disciplinary applications. *Archives of Computational Methods in Engineering* 2010; **17**(3):253–297.
14. Kratos. Available from: http://kratos.cimne.upc.es/kratoswiki [15 December 2010].
15. Oñate E, Valls E, Garcia J. Modeling incompressible flows at low and high Reynolds numbers via a finite calculus-finite element approach. *Journal of Computational Physics Archive* 2007; **224**(1):332–351.

DOI: 10.1002/nme

16. Nadukandi P, Oñate E, Garcia E. A high resolution Petrov-Galerkin method for the 1D convection-diffusion-reaction problem. *Computer Methods in Applied Mechanics and Engineering* 2010; **199**(9–12):525–546.
17. Codina R. Stabilized finite element approximation of transient incompressible flows using orthogonal subscales. *Computer Methods in Applied Mechanics and Engineering* 2002; **191**(39–40):4295–4321.
18. May M, Rossi R, Oñate E. Implementation of a general algorithm for incompressible and compressible flows within the multi-physics code kratos and preparation of fluid-structure coupling. *Technical Report*, CIMNE, Barcelona, 2008.
19. Codina R, Folch A. A stabilized finite element predictor-corrector scheme for the incompressible Navier-Stokes equations using a nodal based implementation. *International Journal for Numerical Methods in Fluids* 2004; **44**(5):483–503.
20. Soto O, Löhner R, Cebral J, Camelli F. Stabilized edge-based implicit incompressible flow formulation. *Computer Methods in Applied Mechanics and Engineering* 2004; **193**(23–26):2139–2154.
21. Bell N, Garland M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM: New York, NY, USA, 2009; 1–11.
22. AMD Accelerated Parallel Processing OpenCL Programming Guide. Available from: http://developer.amd.com/gpu/ATISDK/documentation/Pages/default.aspx [17 December 2010].
23. Donea J, Huerta A. *Finite Element Method for Flow Problems*. Wiley edition: New York, 2003.
24. McCalpin JD. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* Dec. 1995:19–25.
25. Lee VW, Kim C, Chhugani J, Deisher M, Kim D, Nguyen AD, Satish N, Smelyanskiy M, Chennupaty S, Hammarlund P, Singhal R, Dubey P. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*. ACM: New York, NY, USA, 2010; 451–460.

DOI: 10.1002/nme